Dangerously Advanced Python

Mike Verdone
for Protospace, Calgary
May 2009

http://mike.verdone.ca/
Introspection and Metaprogramming

- Introspection: Programs that can interrogate their own data structures and make decisions based on what they find.
- Metaprogramming: Programs that can modify themselves at runtime
Simple Introspection

- Python function `dir` returns a list of attribute names of an object:

```python
>>> dir("hello")
['__add__', '__class__',
'__contains__', '__delattr__',
'__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__',
'__get__', '__getattribute__',
'__ge__', '__getnewargs__',
'__getitem__', '__getnewargs__',
'__get__', '___getslice___', ...
]```
A Bad But Interesting Example

- Get a random method from an object and call it, hope for the best...

```python
def randomCall(obj):
    all_attrs = [
        getattr(obj, attr_name)
        for attr_name in dir(obj)]
    all_methods = [
        attr for attr in all_attrs if callable(attr)]
    some_method = random.choice(all_methods)
    some_method()
```
Python’s Guts are Showing

• Everything is an object (including classes, functions, and methods)

• OO mechanics visible through *magic variables* and dictionaries

• Many of these are assignable...
This is Fun and Dangerous

• Python gives you enough rope to hang yourself
Magic Variables

- Variables or methods surrounded by double underscores are magic
- Modifying them can alter things in very special ways
- At a Python prompt type `help(<type>)` and you will see many of them
  - type can be `int`, `dict`, `str`, etc.
__getitem__

• __getitem__ controls what happens when you do a dictionary lookup on an object: `my_object['key']`

• __setitem__ controls what happens when you set a dictionary item: `my_object['key'] = 'asdf'`
Wikipedia as a Python dict

```python
import urllib2
import json  # need python 2.6 for this library...
import sys

class Wikipedia(object):
    def __getitem__(self, key):
        # All this junk gets the Wikipedia page text via HTTP
        data = urllib2.urlopen(
            'http://en.wikipedia.org/w/api.php?action=query'
            '&prop=revisions&titles=%s&rvprop=content&redirects'
            '&format=json' % (urllib2.quote(key))).read()
        return json.loads(data)['query']['pages'].values()[0] \
            ['revisions'][0]['*'].encode('ascii', 'replace')
```

It looks just like a dict...

```python
wikipedia = Wikipedia()

# Look up 'hacker' and print it to the screen
print wikipedia['hacker']
```
Make Everything Pythonic

• Using magic variables you can make all your interfaces look like normal Python objects

• You can hide method calls and make it easier for other programmers to use your stuff
That was sort of a good idea

The following are more dangerous
Changing a Method at Runtime

class A(object):
    def a_method(self):
        print "hello, I am an A!"

a = A()
a.a_method()  #Calls a_method

def patch_method():
    print "I have been patched"

# Assign the function over the existing method
a.a_method = patch_method

# This calls patch_method
a.a_method()
How does that work?

- Every Python object has a dictionary of attributes
- The dictionary is called \_\_dict\_\_
- Even methods are assignable in this dictionary
a = A()

# First the dictionary is empty
print a.__dict__
==> {}

# After patching the patch method is in the dictionary
a.a_method = patch_method
print a.__dict__
==> {'a_method': <function patch_method at 0x8a7f0>
If the dictionary was empty, how did `a_method` get called?

- If no attribute is found in `__dict__`, the attribute is pulled from the class dictionary.

```
# The class dictionary on the class
A.__dict__

a = A()

# The class dictionary as seen by the object
a.__class__.__dict__
```
The **__class__** variable is also assignable

- That means we can change an object’s class at runtime

```python
class B(object):
    def b_method(self):
        print "I am a B object!"

a = A()
a.a_method()  # This works, a is an A

# Change the class!
a.__class__ = B

a.b_method()  # This works, a is now a B
```
You can also create a class programatically

```python
# Normal way to define a class
class MyClass(object):
    def my_meth(self):
        print "whatever"

# Programatic way to create a class:
def my_meth(self):
    print "whatever"

MyClass = type(
    "MyClass", (object,), {"my_meth": my_meth})
```
Basically...

• Almost everything in Python is exposed as an object

• Almost everything you can modify syntactically can be modified programatically

• Almost nothing is hidden
Abusing the Stack

- Exception objects have a stack which is used to print debugging information
- But you can abuse it and play with the variables in your caller
import sys

def print_x():
    x = ["everything is normal"]
    mess_with_caller()
    print x[0]

def mess_with_caller():
    try:
        raise RuntimeException()
    except:
        e, b, t = sys.exc_info()
        ldict = t.tb_frame.f_back.f_locals
        ldict['x'][0] = "messed up"
Benefits of this stuff

• If your code breaks you can explore it

• You can see how the interpreter is working from the code itself. You don’t need C.

• It’s like seeing the gears turning inside your program.
Keep Exploring

• Use `dir()` on everything
• Use `help()` on everything too
• Set magic variables to strange things, see what happens